Connectivity and Attribute Compression of Triangle Meshes

von Buelow, Max

(2020)

TECHNISCHE
UNIVERSITÄT
DARMSTADT

GCC
Graphics, Capture and
Massively Parallel Computing

Bachelor's Thesis

Connectivity and Attribute Compression of Triangle Meshes

Maximilian Alexander von Bülow

March 2017

Technische Universität Darmstadt
Department of Computer Science
Graphics, Capture and Massively Parallel Computing

Supervisors: Dr. rer. nat. Stefan Guthe
Dr.-Ing. Michael Goesele

Declaration of Authorship

I certify that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged, and has not been submitted, either in part or whole, for a degree at this or any other University.

Darmstadt, 13.03.2017

Maximilian Alexander von Bülow

Abstract

Triangle meshes are used in various fields of applications and are able to consume a voluminous amount of space due to their sheer size and redundancies caused by common formats. Compressing connectivity and attributes of these triangle meshes decreases the storage consumption, thus making transmissions more efficient. I present in this thesis a compression approach using Arithmetic Coding that predicts attributes and only stores differences to the predictions, together with minimal connectivity information. It is applicable for arbitrary triangle meshes and compresses to use both of their connectivity and attributes with no loss of information outside of re-ordering the triangles. My approach achieves a compression rate of approximately 3.50:1, compared to the original representations and compresses in the majority of cases with rates between 1.20:1 to 1.80:1, compared to GZIP.

Keywords: Compression, Data structures, Mesh representation, Algorithms, Graphics

Zusammenfassung

Dreiecksnetze sind in verschiedenen Anwendungsbereichen zu finden und können aufgrund ihrer schieren Größe und Redundanzen, die übliche Formate erzeugen, großen Speicherbedarf verursachen. Kompression der Konnektivität und Attribute dieser Dreiecksnetze verringert den Speicherbedarf und macht die Übertragung effizienter. In dieser Arbeit präsentiere ich einen Kompressionsansatz, der Attribute abschätzt und anschließend die Fehler zu den Schätzwerten zusammen mit minimalen Konnektivitätsinformationen mithilfe eines arithmetischen Kodierers komprimiert. Er ist anwendbar auf beliebigen Dreiecksnetzen und komprimiert Konnektivität und Attribute ohne Verlust von Informationen, mit Ausnahme einer Neuanordnung der Dreiecke. Mein Ansatz erreicht Kompressionsraten von ungefähr 3.50:1 gegenüber der Originalrepräsentationen und komprimiert in den meisten Fällen besser als GZIP mit Raten zwischen 1.20:1 bis 1.80:1.

Titel: Konnektivitäts- und Attributkompression von Dreiecksnetzen

Contents

1 Motivation and Introduction **1**

1.1 Motivation 1

1.2 Introduction 2

2 Related Work **3**

2.1 Comparisons 3

2.2 Connectivity compression 3

2.3 Attribute compression 4

2.4 Implementations 5

3 Background **7**

3.1 Arithmetic Coding 7

3.2 Preliminaries on meshes 9

3.3 Connectivity compression 12

4 Algorithm **15**

4.1 Overview 15

4.2 Cut-Border Machine extensions 16

4.3 Non-manifold meshes 18

4.4 Attribute compression 20

5 Results and Discussion **23**

5.1 Evaluation 25

5.2 Comparison against existing implementations 28

5.3 Graphical inspector 29

6 Conclusion **31**

6.1 Future work 31

Bibliography **35**

1 Motivation and Introduction

1.1 Motivation

Triangle meshes are used in very different areas of application, e.g. 3D rendering, reconstructing and printing, CAD modeling or physical simulations. They can become very space consuming for high resolutions and come with the additional space consumptions from lots of redundancies caused by common mesh formats. Their total space consumption can be for example thousands times higher compared to the corresponding raw texture data, which nevertheless can be compressed very easily as they can be stored using state of the art image formats. In order to address all this areas of applications and solve the former explained problems, developing a general compression algorithm of high effectiveness is mandatory.

Simple mesh formats store the attributes in either binary or ASCII format by concatenating them into simple lists. The connectivity is usually stored using an indirect indexed approach, which uses integer numbers as indices referencing to the vertices. Because of the local similarity of attributes and the global indexing used for the connectivity, this is a very redundant way of storing meshes.

A very common way to compress meshes is to use dictionary coders like GZIP to reduce this simple mesh formats to about the half size. The problem with this method is, that GZIP treats all data the same way, making no differences between connectivity and different types of attributes. In order to improve compression rates, more specialized algorithms need to be developed. Other existing compressors either suffer in terms of generality, ineffective connectivity compression algorithm or bad attribute prediction schemes. To be precise in terms of generality, some compressors enforce quantization of attributes and no current compressor supports arbitrary attributes.

1.2 Introduction

The algorithm proposed in this thesis makes very few assumptions about the input mesh, in order to be able to address as many areas of application as possible. Due to these few assumptions and no further parameters, the algorithm is highly integrable into existing applications. Most applications working on triangle meshes already implement data structures for adjacency lookups, that are as well required by the algorithm I present in this thesis. The only effort of integrating comes from combining the data structures of these applications with the data structures of my algorithm efficiently.

There are no restrictions for the type and size of any attribute, as long as they are either a combination of integer or real numbers and equal in size for each primitive during compression. Currently, the algorithm is restricted to the compression of triangles, but it could easily be extended to encode arbitrary polygons (Section 6.1). To make the algorithm as general as possible, it makes no assumptions about the required precision of the attributes and compresses everything losslessly. In terms of geometry and connectivity, the algorithm I present does not make any changes to the mesh outside of re-ordering the primitives, vertices and attributes. Thus, there is not any loss on adjacency information.

2 Related Work

2.1 Comparisons

In 2005 two complete comparisons of mesh compression approaches were published by Alliez and Gotsman [AG05] and Peng et al. [PKK05]. A decade later, in 2015, the work of Maglo et al. [Mag+15] gave an updated overview about mesh compression in the fields of mesh geometry, attributes and connectivity. Their work defines and compares existing approaches and also yields trends for the future.

Another related work is the one of Limper et al. [Lim+13], which compares different compressed and uncompressed mesh representations against decompression and network transmission time on different types of devices. It also shows that on mobile devices with very limited computational resources, some algorithms are taking more time to decompress meshes than receiving the uncompressed mesh through the network.

2.2 Connectivity compression

Traditional mesh formats like PLY, OBJ or OFF use vertex index lists representing the connectivity information indirectly. Because this type of storing connectivity information is very redundant, different connectivity algorithms were established in the past decades, which can be distinguished into two categories: the single-rate and the progressive ones. The operations generated by this algorithms can be passed to an entropy coder like the Arithmetic Coder described in Section 3.1.

Single-rate Single-rate algorithms are compressing the mesh successively by traversing it. The mesh is divided by a cyclic border into an inner part, which is already compressed and an outer part, which is outstanding for compression. This cyclic border is traversed and extended at each iteration step. The current element on the border, where the extension takes place, is called the *gate*. Single rate algorithms can be distinguished into valence based and triangle based, which both encode the connectivity with a rate

of approximately two bits per vertex, although valence based are slightly more effective than triangle based algorithms.

Touma and Gotsman [TG98] describes a valence based algorithm, which defines the cyclic border as a succession of vertices, called *Vertex Cycle*. The algorithm iterates over the whole triangle fan of the gate vertex and encodes the valence of the not so far compressed vertices.

On the other hand, the algorithm of Gumhold and Straßer [GS98] and Gumhold [Gum99], called Cut-Border Machine is triangle based and defines the cyclic border as a succession of edges, called *cut-border*. While iterating, the algorithm encodes different relations between the gate edge and the cyclic border. The algorithm of Rossignac [Ros99] is very similar and calls the cyclic border *Boundary*. Gumhold et al. [GGS99] are showing an implementation of the Cut-Border Machine for tetrahedral meshes.

Progressive Progressive algorithms, described by Hoppe [Hop96], incrementally simplify the mesh into different levels of details. They encode a basic simplified mesh of the smallest level of detail and refine the mesh by encoding operations like *vertex split*, *edge split* or *face split* for each level of detail, which generate a more detailed surface at the split position by adding more triangles. Progressive algorithms are suitable when different resolutions of the mesh are required. E.g. when a object is rendered in the scene background, a lower resolution looks visually the same and it is sufficient to omit higher levels of detail. Attributes of progressive algorithms are often quantized to around 12 bits [Mag+15]. Progressive meshes are not further discussed in this thesis, due to their lossy encoding of both attributes and connectivity in all, except the highest, levels of details and the less compression rates compared to single-rate algorithms.

2.3 Attribute compression

As described by Maglo et al. [Mag+15] the attributes are first quantized in the case of lossy compression and then, in both lossy and lossless cases, they are delta coded against a predicted attribute.

Prediction The main idea of predicting attributes is exploiting connectivity information to make use of their locality and afterwards encode the differences to the original values (*delta coding*). Adjacent triangles or vertices should have similar attributes, so that differences can become very small. Prediction schemes are further described by Deering [Dee95], Touma and Gotsman [TG98], and Gumhold et al. [GGS99].

The simplest prediction is to assume that the attribute of the vertex is the same as the one of the adjacent vertex. However, a more advanced prediction is to estimate the adjacent vertex using a parallelogram. This approach gives better results because two pairs of adjacent triangles often form approximately the shape of an parallelogram.

Quantization Quantization, described by Deering [Dee95], is done by subdividing a range of integer or real, i.e. floating point, numbers into different cells identified by unique integer numbers. In the case of a quantization of 3D positions, the range of the positions can be seen as the Axis Aligned Bounding Box of the mesh. After subdivision, all attributes are transformed into their respective cell identifier. This approach of attribute compression is lossy, due to the grouping into a small amount of cells and therefore not used in my implementation but further discussed in Section 6.1.

2.4 Implementations

There are a couple of fully featured compression implementations that have been released in the past years. In this section I want to give a short overview of these different implementations and show whether their compression is either lossy or lossless. I also show, whether the implementations are able to compress non-manifold meshes and explain which types of attributes they support. All of them use either an entropy or a dictionary coder, which is called the *compression backend*, to encode the collected connectivity information and the attribute data.

OpenCTM OpenCTM was released by Geelnard [Gee10] in 2009. It uses the dictionary coder LZMA as its backend and the connectivity is encoded using a delta encoded vertex index list. OpenCTM makes no predictions on the attributes but allows optional lossy compression by using quantization with user defined precision. This format is limited to encode positions, texture coordinates and normals as per-vertex attributes in 32 bit floating point precision.

WebGL loader The WebGL loader, developed by Google [Goo11] in 2011, is part of the Google Human Body Project and does not make use of any compression backend. In practice it is meant to be combined with GZIP because it produces UTF-8 output, which can be easily interpreted by JavaScript applications. The vertices are transformed using a vertex cache and connectivity is encoded using a delta coded vertex index list to the transformed vertices. All attributes are quantized, vertex positions are predicted using

a parallelogram and normals using the cross product of the edges. WebGL loader is able to encode positions and normals. Because of the enforced quantization, this format is always lossy.

Open3DGC The Open3DGC project by AMD [AMD13], which was developed in 2013 is an implementation of the work of Mamou et al. [MZP09]. The connectivity is compressed, by rotating the vertex indices of each triangle in a way, that the vertex in the center of each triangle fan will only be encoded once. This is a similar but easier approach to the valence based algorithms. It uses an arithmetic coder as its backend, the geometry data is predicted using a parallelogram and the attributes are quantized. Open3DGC supports a variable number of floating point and integer attributes. Because of the quantization, this format is, as the former implementation, lossy.

Google Draco Google Draco was released by Google [Goo17] in early 2017 and uses the entropy coder ANS [Dud13], which is a combination of Arithmetic Coding and Huffman Coding, as its backend for attribute data and static Huffman Coding for connectivity. The connectivity is compressed using the Edgebreaker [Ros99] algorithm. Positions are predicted using multiple parallelograms, texture coordinates are predicted using the vertex positions and all other attributes are delta coded. Draco supports compressing vertex positions, colors, normals and texture coordinates but cannot encode non-manifold meshes without losing adjacency information. The loss of adjacency information is due to the inability of the used Edgebreaker algorithm to encode non-manifold meshes. To fix this issue, Draco copies vertices and its attributes on every non-manifold edge or vertex to ensure a 2-manifold mesh.

3 Background

In this chapter I describe the basic concepts used for compressing triangle meshes. This thesis is structured as follows: Section 3.1 describes a method to encode a stream of symbols and a way for efficient handling of cumulated frequencies. Section 3.2 gives an overview about meshes, shows how they are defined and consist of and describes a data structure for efficient adjacency lookup. Finally, in Section 3.3, I present an algorithm for compressing triangle mesh connectivity.

3.1 Arithmetic Coding

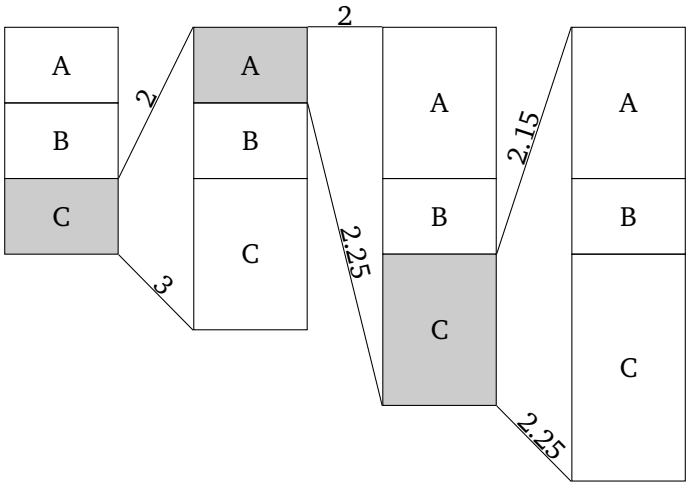


Figure 3.1: Arithmetic Coder with an adaptive frequency model encoding the character sequence *CAC*.

The Arithmetic Coder is a lossless entropy coder, which is able to archive nearly optimal compression rates, i.e. the size of the result is nearly the size of the entropy. The main difference to Huffman Coding is, that coded symbols are not required to have an integral amount of bits, which enables a better compression rates. The main idea is to subdivide an interval into different parts representing the alphabet and periodically reducing it to

the part representing the current symbol. Major challenges are to efficiently implement this interval reduction for computers with limited decimal number precision.

Algorithm According to Rissanen and Langdon [RL79] a bit stream can be seen as a single fractional number or alternatively as an interval containing all bit streams with the same prefix. The implementation used in this thesis is based on the work of Moffat et al. [MNW98]. I chose Arithmetic Coding because it archives better compression rates than other entropy or dictionary based compression algorithms, despite of its comparably worse compression speed.

Given a symbol s_i, a cumulated frequency range $r_i = [l_i, u_i)$ exists. Also an interval $[L, L+R)$, describes the current state of the range coder, where initially $L = 0$, $R = t$ and t is the sum of all frequencies u_n. After encoding symbol s_j, the algorithm reduces this interval to $L' = L+R/t \times l_j$ and $R' = R/t \times (u_j - l_j)$. When all symbols are processed, the algorithm writes an arbitrary number from the interval $[L, L+R)$, e.g. L, to the output stream.

Theoretically a high number of bits is required to represent L and R correctly, which makes the reduction of the interval very hard and inefficient. To solve this issue, the interval is represented using fixed precision numbers and renormalized repeatedly. Renormalization of the interval keeps R in the range $(2^{b-2}, 2^{b-1}]$, where b is the number of bits chosen to represent L and R. When $R \leq 2^{b-2}$, the interval $[L, L+R)$ must be scaled by halving it. Additionally, if $L + R \leq 2^{b-1}$, the algorithm writes a zero to the output stream and if $L \geq 2^{b-1}$, it writes an one and moves L 2^{b-1} left before scaling the interval. If none of this cases happen, for the interval applies $2^{b-1} \in [L, L+R)$, the algorithm remembers the number of consecutive occurrences and appended such many opposite bits of the following one or zero, i.e. it writes a value slightly above or below 0.5 in such case.

The cumulated frequencies are maintained by the so called *model*. In our case, the model simply counts the frequencies of each symbol and outputs the cumulated frequency. The later described connectivity compression uses conditional frequencies, which requires multiple sub-models for each condition. The combination of encoding and maintaining an adaptive model is demonstrated in Figure 3.1. The basic idea behind any model is that it tries to predict the next symbol's probability as efficient as possible in order to maximize the compression rate.

Managing cumulated frequencies Since the extraction of cumulated frequency happens at each encoding step, the implementation of Moffat et al. [MNW98] focuses on

optimizing it. A naive implementation would require n steps for calculating the cumulated frequencies by iterating through the frequency values and adding them. The reverse lookup, which is required for decompression, also requires n steps to find the symbol for a given cumulated frequency. However, an optimization of the extraction of cumulated frequencies can be applied by managing an array of the cumulated frequencies directly, which enables constant time extraction. Again, this solution requires n steps for updating a frequency, which makes this a useless optimization.

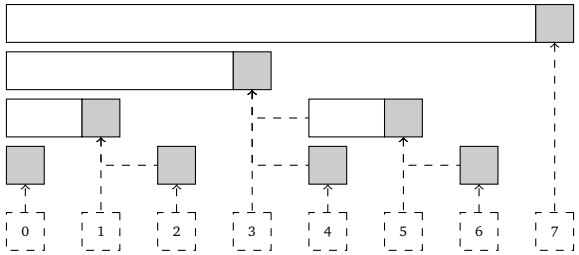


Figure 3.2: Visualization of the Fenwick Tree [Fen93] and the traversal of it.

To solve this problems and realize fast extraction, update and reverse lookup of cumulated frequencies, Moffat et al. [MNW98] implemented the Fenwick Tree [Fen93], that stores the frequencies of each symbol in a binary tree as can seen in Figure 3.2. This way, logarithmic time is archived for extracting and updating frequencies [Fen95] in both directions. In addition, it possesses, linear storage complexity, like the former presented techniques.

3.2 Preliminaries on meshes

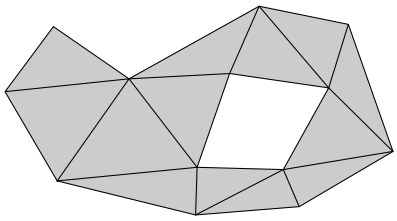


Figure 3.3: A triangle mesh.

According to Maglo et al. [Mag+15], a mesh consists of geometry, connectivity and attributes. Due to the structure of my algorithm I make no distinction between the geometry and the attributes. The geometry can always be seen as the position attribute of a vertex and can be optional as well as described by Isenburg et al. [IGG01].

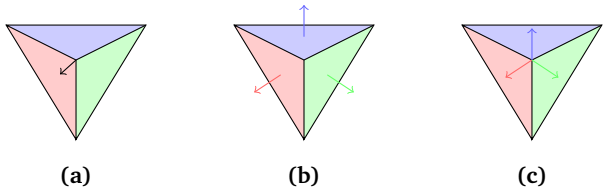


Figure 3.4: (a) A vertex attribute, (b) triangle attributes and (c) triangle-vertex attributes demonstrated using different types of normals.

An attribute can be seen as a vector of real or integral scalar numbers, that can be either empty, i.e. has a dimension of zero, or not. For example, the position p of a vertex is a vector (p_x, p_y, p_z). Multiple attributes that always occur in a union can be grouped together. For example, if the mesh has no hard edges, the vertex normals n are always bound to the vertex positions and form the attribute $(p_x, p_y, p_z, n_x, n_y, n_z)$. Attributes with the same signature form a series, called *attribute list*.

The series of vertices is represented by exactly one attribute list, called *vertex attributes*. Triangles are formed by another attribute list, called *triangle attributes*, and the combination of three indices to the vertex list for every corner point of each triangle. All other attribute lists, called *triangle-vertex attributes* are bound to the corners of triangles and can be shared between different vertices or triangles. Edges can have attributes attached to them as well, but they will not further be discussed in this thesis due to their rare use. The different types of attributes can be seen in Figure 3.4. The connectivity of the mesh can be either represented *directly* by storing all neighbors for each triangle or *indirectly* using the vertex indices at the corners of each triangle. In the latter approach all triangles that share a pair of vertex indices, called *edge*, are adjacent to each other.

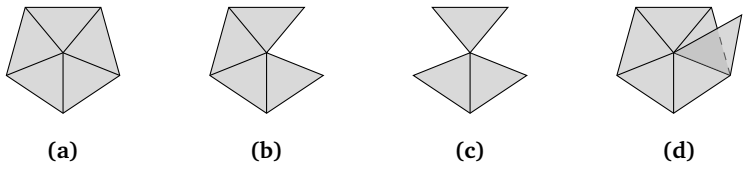


Figure 3.5: 2-Manifold and non-manifold triangle fans: (a) a closed triangle fan (2-manifold), (b) an open triangle fan (2-manifold), (c) a not opened/closed triangle fan (non-manifold) and (d) more than one adjacent edge (non-manifold).

The mesh connectivity can be seen as the topology of the mesh, which forms the surface of it as visualized in Figure 3.3. As it can be seen in Figure 3.5, surfaces are

2-manifold when all triangles of a vertex attached to it form an either closed or open triangle fan. When this is not the case, a vertex is connected to multiple triangle fans and the surface is called non-manifold.

Data structure for triangle connectivity Most mesh formats, like PLY, OBJ or OFF, which are assumed as the input mesh formats, are storing the triangle connectivity information using the indirect approach, which stores indices for each corner of a triangle. This way of storing connectivity information requires a linear-time lookup for finding an adjacent triangle because the whole triangle list has to be traversed for it. Since connectivity compression algorithms make aggressive use of adjacency information, an efficient direct connectivity data structure should be used to allow faster access.

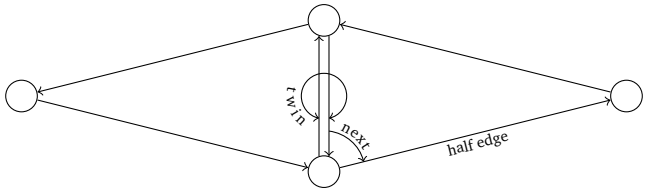


Figure 3.6: Visualization of the Half Edge Data Structure.

An efficient direct connectivity data structure to allow constant-time lookup of adjacent triangles is the Half Edge Data Structure [CKS98]. A *half edge* is a directed edge within a triangle and the direction is defined by the order of the indices to the vertex attribute list described above. The Half Edge Data Structure maintains a *next* pointer for each half edge referencing to the next half edge of the triangle forming a circle of three edges. Each edge, which is not on a mesh border, has a *twin* pointer attached to it, which is pointing to the reverse edge, i.e. the edge of the adjacent triangle. A visualization of this can be seen in Figure 3.6. Vertex attributes can be referenced from the beginning of each half edge and triangle attributes can be referenced from each half edge of the triangle. The Half Edge Data Structure comes with the problem, that each half edge can only be bound to exact one triangle, which makes non-manifold edges unrepresentable. Nevertheless, most triangle meshes have some non-manifold edges.

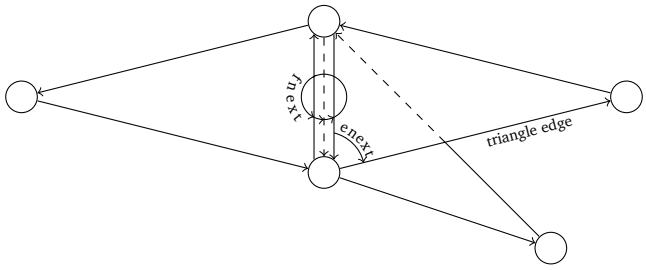


Figure 3.7: Visualization of the Triangle Edge Data Structure.

In order to also support non-manifold meshes, I chose the Triangle Edge Data Structure [Müc93], which extends the Half Edge Data Structure. In this data structure, the half edges form a tuple with their triangle, called *triangle edge*, and reference their adjacency using a ring with all half edges equal or reverse to it. This is done by maintaining the twin pointers in a way that they are referencing all adjacent Triangle Edges in cyclic way. The next pointer is here called *enext* and the circular twin pointers are called *fnext*. The Triangle Edge Data Structure is visualized in Figure 3.7.

3.3 Connectivity compression

The Cut-Border Machine of Gumhold and Straßer [GS98] and Gumhold [Gum99] is used to compress triangle mesh connectivity. By traversing the mesh in a structured way and writing only information required for correct traversal during decoding, nearly optimal compression can be performed.

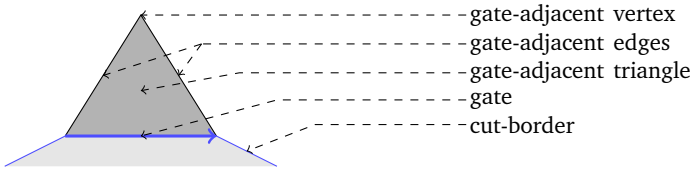


Figure 3.8: Terms of the Cut-Border Machine using the example of the *new vertex* operation.

The main part of the Cut-Border Machine is the so called *cut-border*. The cut-border splits the mesh into an encoded and decoded part and is stored as a circular data structure, that contains the half edges that are currently on this circle. For each iteration, the Cut-Border Machine stores a reference to the current cut-border element, the so called gate. The gate leads from the compressed to the uncompressed region and is the only part of the cut-border where new triangles are appended to the cut-border. In the following, I define multiple terms (Figure 3.8) for better understanding of the algorithm. I

call the triangle of the twin of the gate *gate-adjacent triangle* and the vertex of the gate-adjacent triangle, which is actually not connected to the gate, *gate-adjacent vertex*. The two edges of the gate-adjacent triangle, which are not the gate, are called *gate-adjacent edges*.

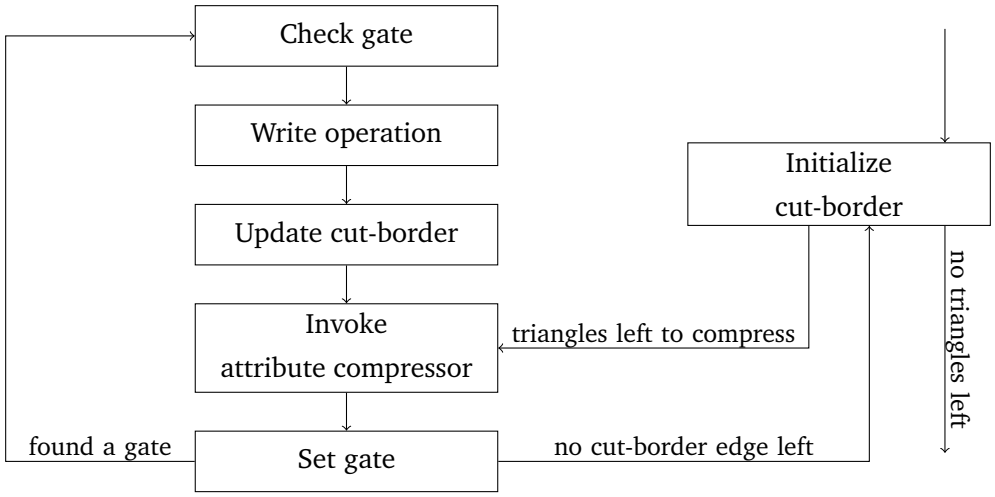


Figure 3.9: Flow chart of the Cut-Border Machine.

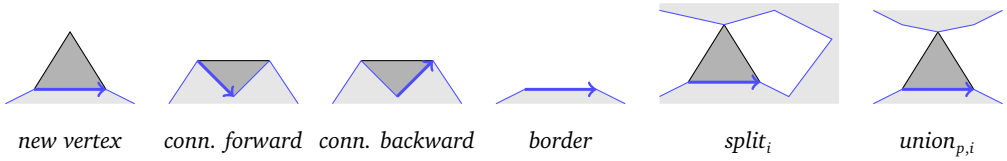


Figure 3.10: Cut-border operations [Gum99]. The current cut-border is marked blue, the gate is marked as a bold blue arrow and the gate-adjacent triangle is shaded dark grey.

Traversal At the beginning, the Cut-Border Machine chooses a triangle from the set of triangles, initializes the cut-border to the three edges of it and sets the gate to its first edge. Now, the Cut-Border Machine checks the gate-adjacent vertex, which can meet the following conditions visualized in Figure 3.10.

The first case is that this vertex simply does not exist because the gate is on a mesh border. For this case a *border* operation will be written and the cut-border edge of the gate will be deactivated. If the gate-adjacent vertex is not part of the cut-border, the gate-adjacent edges will be added to the cut-border and a *new vertex* operation will be written.

The last case is, that the gate-adjacent vertex is part of the cut-border. This behavior can be split again into multiple different sub-cases. If the vertex is the next vertex on the cut-border after the gate, the gate will be directly connected to the edge after the next element and a *connect forward* operation will be encoded. Analogous, a *connect backward* operation will be encoded, when the gate-adjacent vertex is the previous one. If the vertex is as well not the previous as the next on the cut-border, the cut-border must be split up into two *parts*, the part in traversal direction from the gate to the adjacent vertex and the part in opposite direction. This will be encoded as a *split*, or alternatively called *connect*, operation, followed by the offset of the adjacent vertex on the cut-border. Due to the splitting of the cut-border, a new condition can occur. When the adjacent vertex is part of another cut-border, they have to be concatenated at this vertex using the *union* operation, followed by the offset and the part index. After each operation, the attribute encoder is invoked to encode the attributes of the current triangle, vertices or combinations of it.

After checking the conditions, updating the cut-border and encoding the cut-border operations, a new gate is determined in a way that a breadth-first search is performed. This happens by choosing the new gate as the following edge of the gate. Regardless of that, in Section 4.2 will be described, that a depth-first search archives better compression efficiency in combination with a Arithmetic Coder [Gum99]. When there is no edge left on the cut-border, the Cut-Border Machine tries to find a distinct edge connected component of the mesh and repeats the previous steps. If all edge connected components are encoded, the Cut-Border Machine stops. The whole algorithm flow is visualized in Figure 3.9.

4 Algorithm

In this chapter I describe the algorithm of a fast triangle mesh compressor that is able to encode arbitrary non-manifold triangle meshes with arbitrary structured attribute data.

4.1 Overview

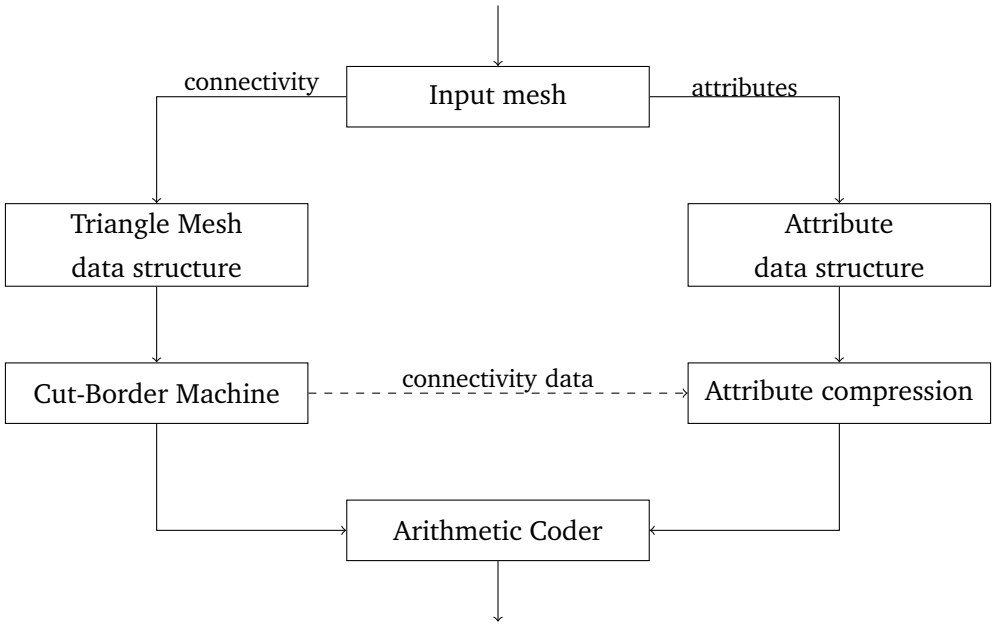


Figure 4.1: Algorithm structure: Data flow of the connectivity and the attributes.

For my algorithm, I chose the Cut-Border Machine (Section 3.3) for connectivity compression because it has very good compression rates and allows easy implementation of extensions (Section 4.2), e.g. to allow compression of non-manifold meshes (Section 4.3). Modification of valence based systems to allow non-manifold meshes would be extremely complicated due to their heavy use of closed triangle fans. The handling of borders is also much more difficult in valence based algorithms, where holes are filled

with a dummy vertex to form a triangle fan. The Edgebreaker algorithm on the other hand requires the entire mesh border to be encoded up front.

Arbitrary attribute data is predicted using adjacency or history information gathered from the connectivity of the mesh (Section 4.4) and delta coded against the prediction. As mentioned above, the mesh connectivity is traversed by the Cut-Border Machine, which also manages binding the attributes to the corresponding vertices or triangles.

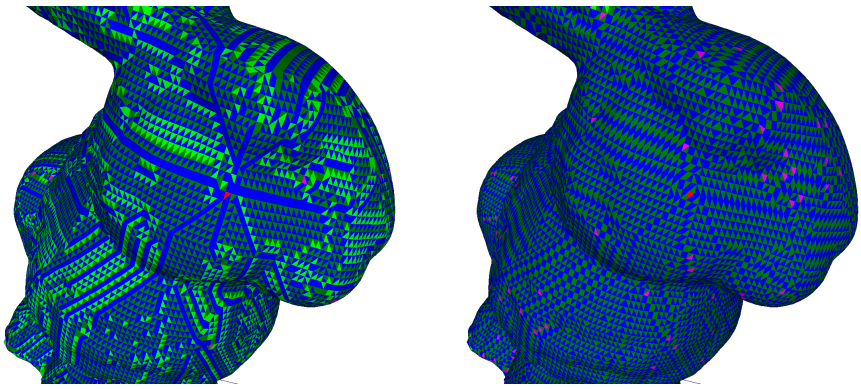
All gathered information, that is, Cut-Border Machine operations, offsets and delta coded attribute data, will finally be compressed using an Arithmetic coder (Section 3.1) as the backend. This could potentially use different output streams for different attributes and connectivity information.

A visualization of the whole algorithm structure described above can be seen in Figure 4.1.

4.2 Cut-Border Machine extensions

I have implemented multiple extensions to the Cut-Border Machine, which are reducing the encoded size of the connectivity.

Arithmetic Coding As described by Gumhold [Gum99], the basic optimization was to encode the cut-border operations using the Arithmetic Coder Section 3.1 with adaptive frequencies for each operation. To make Arithmetic Coding efficient for the Cut-Border Machine several changes have to be applied. The basic idea is to make the compression as effective as possible by having few operations with high frequencies and a majority of operations with very low frequencies. This keeps the number of renormalizations of the interval of the Arithmetic Coder low and increases the coding efficiency.



(a) breadth-first search **(b)** depth-first search

Figure 4.2: Visualization of the (a) breadth-first search and the (b) depth-first search traversal order. Triangles compressed by the new vertex operation are marked blue, connect backward operations light green, connect forward operations dark green, initial operation red and split operations magenta.

Traversal order Gumhold [Gum99] mentions that the number of *connect forward* and *connect backward* operations are almost the same when performing a breadth-first search. As mentioned before, it would be better to find a traversal technique where either *connect backward* or *connect forward* operations have a low frequency. When traversing the mesh in a way that a depth-first search is performed, by choosing the gate as the first newly created cut-border edge, the number of *connect forward* symbols overwhelm the *connect backward* symbols as can be seen in Figure 4.2. This keeps the start vertex of the gate as long as possible static and a traversal around the triangle fan of this vertex happens as long as *new vertex* operations occur. When the last triangle of the triangle fan is about to be encoded, a *connect forward* operation closes the triangle fan and finally changes the start vertex of the gate to traverse a new triangle fan. This traversal order is locally equal to the traversal orders used in valence based systems described in Section 2.2.

Conditional frequencies An observation of Gumhold [Gum99] was, that it is much more likely that a *connect* operation occurs, when the so called order of a vertex, is big and a *new vertex* occurs more likely when the order is small. The order is defined as the current number of encoded faces, which are connected to the given vertex, i.e. the valence of the mesh up to this point. To make the Cut-Border Machine more efficient, conditional frequencies are used for encoding the operations. The different conditions are bound to the order of the vertex, which enables distinct frequency gathering for all

different orders. Due to the low frequencies of all operations, that are not a *new vertex* and *connect backward* operation, the implementation makes no distinction on the orders of those.

Condition	Operations
Cut-border contains no edges	*initial* possible
Cut-border contains edges	*initial* impossible
Previous element on Cut-border was previously encoded as a mesh border	*connect backward* impossible
Next element on Cut-border was previously encoded as a mesh border	*connect forward* impossible
Cut-border is not split	*union* impossible

Table 4.1: Impossible operations and their conditions.

Impossible operations In some cases some operations are impossible to occur as it can be seen in Table 4.1. E.g. a connect backward operation can never happen, when the previously encoded edge was a mesh border. This cases can again be used as conditions for the Arithmetic Model and the frequencies of the corresponding impossible operations can be set to zero. Gumhold [Gum99] however, solves this task by renaming infrequent operations to impossible operations. An observation was, that executing a *connect backward* or *connect forward* operation over an already encoded border just removes the gate which has exactly the same behavior as a *border* operation. In that special case, the border operation can just be renamed to the corresponding *connect* operation to reduce number of *border* operations, which is more efficient in speed than handling conditional frequencies.

4.3 Non-manifold meshes

An important thing the traditional Cut-Border Machine of Gumhold and Straßer [GS98] lacks, is capability of encoding all non-manifold edges and vertices. In this section, I propose a generalized compression scheme, that works for arbitrary input meshes.

The basic Cut-Border Machine [Gum99] can only encode non-manifold vertices on mesh borders and non-orientable vertices which forces the approach to cut meshes into manifold edge connected components while losing adjacency information at the cuts.

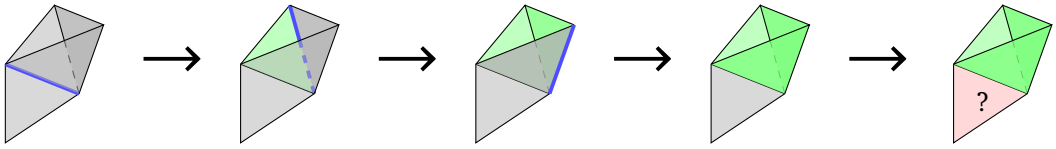


Figure 4.3: Traversal of a non-manifold mesh using the Cut-Border Machine. The current gate is marked blue.

When analysing why the Cut-Border Machine cannot encode non-manifold vertices or edges, I observed that it assumes every gate-adjacent vertex to be located on the cut-border when it was flagged as previously encoded. This is exactly the condition for any *connect* operation. On 2-manifold meshes, this is always the case because each edge is only handled twice during compression – once when it is added to the cut-border and again when it becomes the gate or some gate connects to that edge. During the second encounter, the edge is always removed from the cut-border. On non-manifold meshes, an edge can get connected or become the gate multiple times. Due to the removal of the edge this can no longer be encoded. However the deletion is required to ensure a correct traversal order. The problems caused by the traversal of non-manifold meshes can be seen in Figure 4.3.

My approach solves this task by handling the case where a gate-adjacent vertex was flagged as already encoded but does not exist on the cut-border as a *new vertex* operation. During encoding the global vertex offset is transmitted instead of transmitting the whole vertex attribute as this is already in the stream. Using this technique, the mesh is split into multiple pieces implicitly that are compressed separately but the decoder is able to to concatenate those pieces correctly using the global vertex offset. I call the modified *new vertex* operation *non-manifold new vertex*. Transmitting the global vertex offset is an acceptable efficiency loss, due to the few occurrences of non-manifold vertices or edges. This solution does not need any costly preprocessing steps to recognize non-manifold vertices or edges by analysing all triangle fans.

Similar to Gumhold [Gum99], I further introduced three new *initial* operations to initialize a new edge-connected mesh component with the offsets of one, two or three already encoded (non-manifold) vertices to keep adjacency information.

4.4 Attribute compression

Data structure for attributes As described in Section 3.2, attributes are represented as a series of vectors. To support vectors of variable in size and type, a dynamic data structure must be implemented.

To realize this data structure, without the need of many memory allocations and wasting memory, only one single byte array that will be reinterpreted to the corresponding types of the different attributes during the run-time. The size s of each tuple can be computed in before, by calculating the size of each tuple element. To encode n tuples, $s \times n$ bytes need to be allocated for such an array.

Duplicates In order to achieve high compression rates and to keep all adjacency information, the algorithm avoids encoding any duplicates of attributes.

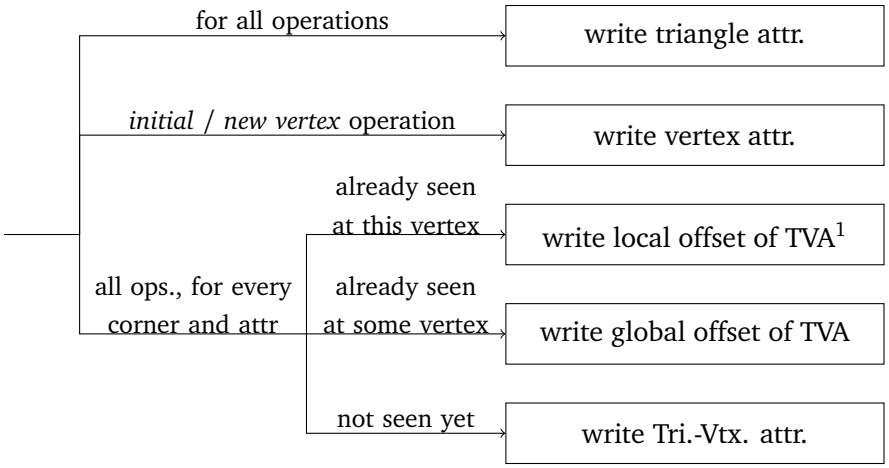


Figure 4.4: Decision tree of the attribute compressor.

Triangle attributes can be encoded after all cut-border operations and vertex attributes can be encoded for each *initial* and *new vertex* operation without any duplicates. Triangle-vertex attributes come with a little more overhead.

Gumhold and Straßer [GS98] describe an algorithm, that subdivides the triangle fan of each vertex into regions with equal triangle-vertex attributes. This is done by saving a reference to the attribute of the *left* and *right* triangle for each cut-border vertex corresponding to a backward and forward traversal order. The references change during the traversal of the cut-border if the cut-border edge is on a crease. Those changes are encoded using control bits for each added edge. Another bit is required to encode

[1] Triangle-Vertex attribute

whether the attribute data must be transmitted or the added edge closes the triangles fan whereby the data can be used from the *right* reference.

This approach comes with the disadvantage in cases where the same triangle-vertex attribute is included in multiple regions, whereby triangle-vertex attributes will be duplicated for each region (*local redundancies*). Additionally, it duplicates triangle-vertex attributes if they are attached to multiple triangles (*global redundancies*).

To avoid local redundancies, my algorithm stores references of all already encoded triangle-vertex attributes for each vertex into a linked list, which I call *local history*. When the algorithm encodes a triangle-vertex attribute, it searches the attribute index in this list and only encodes the *local offset* to it. If the attribute was not found, it is added the local history. Additionally, to avoid global redundancies, the algorithm encodes the *global offset* to the *global history* of attribute in case it is already encoded by a previous vertex. The entire behavior of attribute coding can be seen in Figure 4.4.

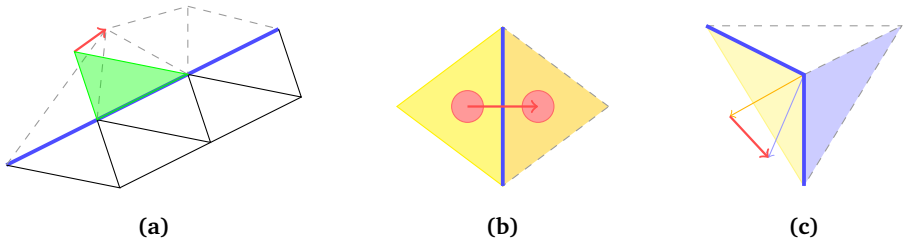


Figure 4.5: Predictions of (a) a vertex position using a parallelogram, (b) a triangle color using the difference to its neighbor and (c) a triangle-vertex normal using the difference to an already encoded attribute of the vertex. The difference is marked as a red arrow and the cut-border is marked in blue.

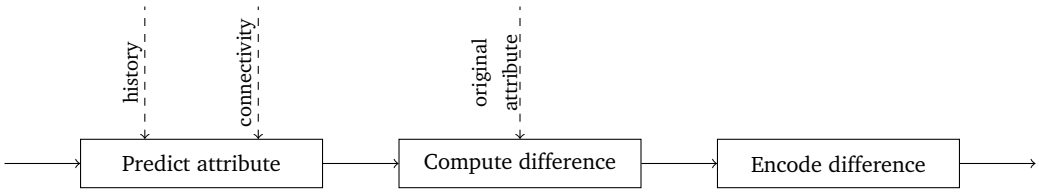


Figure 4.6: Flow chart of attribute prediction.

Prediction To keep the number of distinct symbols low, I decided to encode only differences to a predicted attribute [Mag+15] as can be seen in Figures 4.5 and 4.6. In order to achieve maximum coding efficiency, the differences should require as few bits

as possible and should be similar to each other. The prediction algorithms need to be different for each attribute type.

- Vertex attributes are predicted using a parallelogram to the opposite gate-adjacent vertex [TG98; GGS99].
- Triangle attributes are encoded as the difference to one of their neighbors.
- Triangle-vertex attributes are encoded as the difference to a former encoded attribute of that vertex.

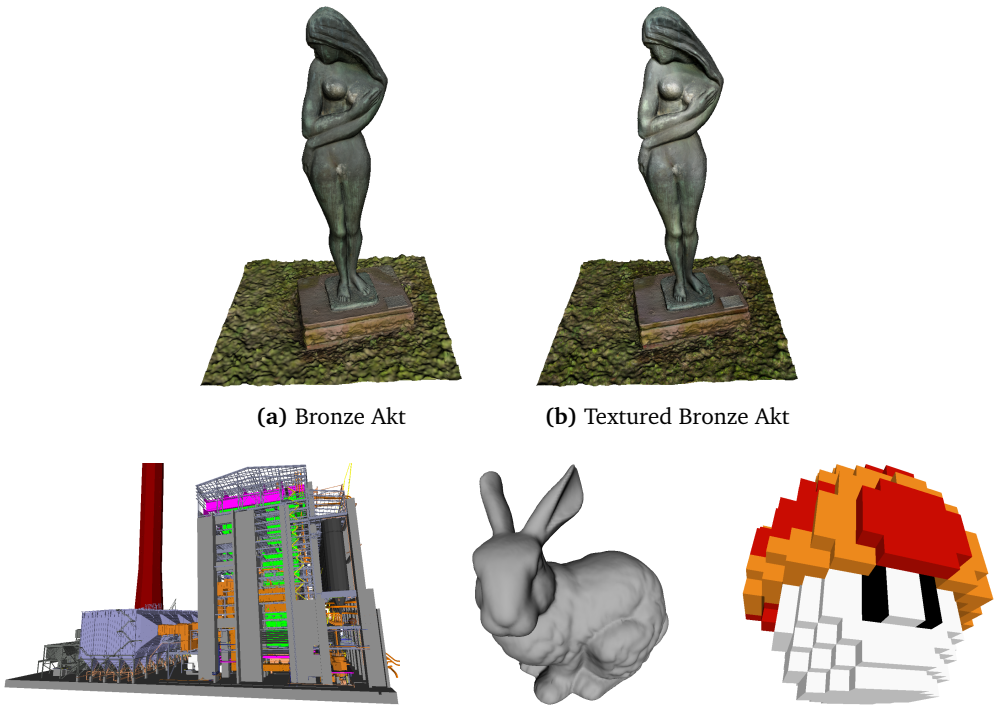
Because computing differences in floating point arithmetic suffers from loss of accuracy, a transformation to a signed integer representation happens that should be orderable according to the ordering of the number it represents. An observation from floating point numbers is, that the concatenation of the exponent and mantissa is always already ordered because of their partial ordering and the higher significance of the exponent. Thus, to allow integer arithmetic ordering of the whole floating point number, inclusive the sign bit, the bits of the exponent and mantissa must be flipped, when number is negative, which was described by Lindstrom and Isenburg [LI06]. Positive and negative infinity representations are also ordered correctly due to their exponent, which is chosen to be greater than all others. This keeps differences between a positive and negative number, as expected, close to zero.

Because the Arithmetic Coder can only handle unsigned integer values efficiently, an additional transformation from signed to unsigned integers is needed. This is done by rotating the bits of the signed integer representation left and inverting all bits, in the case of a negative number. After this transformation, the signed integer numbers with the same absolute numbers are arranged side by side, which makes their difference is as small as possible. Google calls this mechanism ZigZag encoding for their ProtoBuf application but it was initially described by Elias [Eli55] in a less formal way.

5 Results and Discussion

I evaluated my algorithm on a variety of different 3D models. The models vary in their:

- origin (modeled, 3D reconstruction, etc.)
- attribute count and type
- triangle and vertex count
- manifoldness



(a) Bronze Akt **(b)** Textured Bronze Akt

(c) Power Plant **(d)** Stanford Bunny **(e)** Mushroom

Figure 5.1: Renderings of the models.

Bronze Akt The Bronze Akt (Figure 5.1a) is a 3D reconstruction using the MVE pipeline [FLG14]. Its vertex attributes are $(p_x, p_y, p_z, c_r, c_g, c_b, x, s)$, where p is the position as 32 bit floating point numbers and c the color value as 8 bit integer numbers. x is the vertex confidence and s the scale of the vertex, both in 32 bit floating point representation. It contains 3.76 millions of vertices and 7.52 millions of triangles. This model contains non-manifold vertices and edges.

Textured Bronze Akt The textured [WMG14] version of the same Bronze Akt model (Tex Akt, Figure 5.1b). Its vertex attributes are (p_x, p_y, p_z) and it has two groups of triangle-vertex attributes, one for representing normals (n_x, n_y, n_z) and one for the texture coordinates (t_u, t_v). All values are real numbers with no specified data type due to the ASCII representation, which are internally handled as 32 bit floating point values. This model is, like the Bronze Akt model, non-manifold.

Power Plant The Power Plant model (Figure 5.1c) is published by the University of North Carolina and has $(p_x, p_y, p_z, n_x, n_y, n_z, c_r, c_g, c_b)$ as its vertex attributes and (c_r, c_g, c_b) as its face attributes. It is a modeled dataset containing 11.07 millions of vertices and 12.75 millions of triangles. This model contains a bigger amount of non-manifold vertices and edges compared to the previously presented models.

Stanford Bunny The Stanford Bunny (Figure 5.1d), published by the Stanford University, is a small model reconstructed from a 3D scanning. Its vertex attributes are (p_x, p_y, p_z, x, i), where p is the position and x and i is the vertex confidence and the intensity, all represented as 32 bit floating point numbers. It contains $35.95k$ vertices and $69.45k$ triangles and is 2-manifold.

Mushroom The Mushroom (Figure 5.1e) is a triangulation of a voxel mesh with (p_x, p_y, p_z) as its vertex attributes and $(n_x, n_y, n_x, c_r, c_g, c_b)$ as its triangle attributes representing the normal and color of a triangle. The attributes are as well represented as 32 bit floating numbers except the color, which is represented as 8 bit integer numbers. The mesh contains 1306 vertices and 2672 triangles and contains non-manifold edges due to its voxel structure.

5.1 Evaluation

Model	$s_{original}$	s_{my}	s_{gzip}	r_{my}	r_{gzip}	$r_{gzip:my}$
Bronze Akt	184.21M	51.93M	93.71M	**3.55:1**	1.97:1	1.80:1
Tex Akt	414.68M	102.86M	173.71M	**4.03:1**	2.39:1	1.69:1
Power Plant	502.88M	145.69M	118.87M	3.45:1	**4.23:1**	0.82:1
Bunny	1622.05k	473.02k	816k	**3.43:1**	1.99:1	1.73:1
Mushroom	90.81k	13.92k	16.54k	**6.53:1**	5.49:1	1.19:1

Table 5.1: Comparison against GZIP, where s is the total size in bytes and r is the compression rate.

Compression rates As it can be seen in Table 5.1, my algorithm compresses with rates between 3.43:1 and 6.53:1 on the selected models. Compared to GZIP, my algorithm compresses with a rate of between 1.69:1 and 1.80:1 on scanned or reconstructed models and with a worse rate of between 0.82:1 and 1.73:1 on synthetic models. The Power Plant model is the only model, which has worse compression rates on my algorithm than GZIP. This is due to the structure of the mesh that contains very few different normals because of many perfectly planar surface parts. Normals are replicated for each vertex and attached to it. The Cut-Border Machine traverses the mesh independently from the surface parts, which results in encoding many different delta values, although it has only few different normals. GZIP can handle this better because it does not do any predictions and difference calculations on the normals. The problem can be solved by storing the topology properly by assigning the normals as triangle-vertex attributes instead of vertex attributes.

	Conn.	Vertex attributes				Tri. attr.		Tri.-Vtx. attr.	
Model	$\sim_c$	$\sim_{pos}$	$\sim_n$	$\sim_{col}$	$\sim_{other}$	$\sim_n$	$\sim_{col}$	$\sim_n$	$\sim_{tex}$
Br. Akt	2.29%	45.51%		8.43%	43.77%				
Tex Akt	1.26%	25.34%						41.18%	32.22%
Pw. Pl.	2.67%	43.57%	49.07%	2.41%			2.27%		
Bunny	1.39%	59.20%			39.41%				
Mush	6.44%	25.16%				60.81%	7.59%		

Table 5.2: The distribution ($\sim$) of connectivity and each attribute.

Due to the strong variation of attributes, I measured how the compression performs on different types of attributes. As mentioned in Section 4.1, multiple output streams can be used for different attributes or connectivity, so the compressed size can be measured

separately. The distribution on the compressed data of each attribute can be seen in Table 5.2.

	Conn.	Vertex attributes				Tri. attr.		Tri.-Vtx. attr.	
Model	r_c	r_{pos}	r_n	r_{col}	r_{other}	r_n	r_{col}	r_n	r_{tex}
Br. Akt	75.89:1	1.91:1		2.58:1	1.32:1				
Tex Akt	75.84:1	1.89:1						3.49:1	4.00:1
Pw. Pl.	39.28:1	2.09:1	1.86:1	9.45:1			11.54:1		
Bunny	126.49:1	1.54:1			1.54:1				
Mush	36.07:1	4.51:1				3.82:1	7.65:1		

Table 5.3: Compression rates of connectivity and attributes. r is the compression rate of the connectivity data (r_c), vertex position (r_{pos}), normals (r_n), colors (r_{col}) and texture coordinates (r_{tex}).

	Conn.	Vertex attributes				Tri. attr.		Tri.-Vtx. attr.	
Model	bpv_c	bpv_{pos}	bpv_n	bpv_{col}	$bpv_{oth.}$	bpt_n	bpt_{col}	$bptv_n$	$bptv_{tex}$
Br. Akt	2.53	16.77		3.11	16.12				
Tex Akt	2.53	50.79						13.76	10.76
Pw. Pl.	2.81	15.29	17.22	0.85			2.08		
Bunny	1.47	31.15			20.74				
Mush	5.45	21.29				12.57	1.57		

Table 5.4: Amount of bits required to encode the connectivity or attributes of a vertex (bpv), a triangle (bpt) or a triangle-vertex ($bptv$) pair.

As can be seen in Table 5.4, the connectivity can be compressed with a rate of between 1.47 and 2.81 bpv, except for the Mushroom which requires 5.45 bpv. This is due to its very huge amount of *non-manifold new vertex* operations compared to the total mesh size that come from the voxel structure of the mesh. Another similar observation is that the Stanford Bunny, which is a 2-manifold mesh, requires the fewest bits per vertex. This is due to the additional cut-border operation introduced and the extra cost of encoding global vertex offsets. The bpX values are computed as follows: $1bpv = \#bits/\#vertices$, $1bpt = \#bits/\#triangles$ and $1bptv = \#bits/(\#triangles * 3)$, where $\#bits$ is the amount of bits required to store the series of attributes divided by the dimension of the attribute vectors and $\#triangles \approx 2 \times \#vertices$ for the selected models.

The vertex position, which is always predicted using a parallelogram, is compressed with a rate of between 1.50:1 and 2.10:1, as it can be seen in Table 5.3. Again, the Mushroom dataset is the only exception with a rate of approximately 4.50:1 because

the parallelogram prediction is perfectly correct for most cases, again due to the voxel structure. It can also be observed, that colors can be compressed with a rate of approximately 2.5:1 for the real world *Bronze Akt* dataset and with a rate of about 10:1 for synthetic models, due to their very static coloring compared to the real world models. Normals are compressed with a rate similar to the positions. Other attributes generated from 3D scanning or stereoscopic 3D reconstruction, are compressed with a rate of about 1.40:1 to 1.70:1.

	Compression			Decompression	
Model	t_{init}	t_{my}	t_{gzip}	t_{my}	t_{gzip}
Bronze Akt	8.39*s*	**11.87s**	33.60*s*	8.24*s*	**2.04s**
Tex Akt	32.79*s*	**28.24s**	176.81*s*	24.66*s*	**5.48s**
Power Plant	16.33*s*	**29.31s**	102.75*s*	25.3*s*	**3.21s**
Bunny	215*ms*	**117ms**	1.06*s*	126*ms*	**51ms**
Mushroom	14*ms*	**5ms**	39*ms*	4*ms*	**3ms**

Table 5.5: Timings of the initialization and the compression and decompression algorithm on a *Intel Xeon E5-2650 v2* CPU.

Compression performance To measure time, I left out the IO delays for both GZIP and my algorithm and measured the time of parsing the mesh and building the adjacency data structure separately from the actual compression. I did so, as most applications are already using such a structure internally, which can be used directly with my compressor. Additionally, while decompressing, my algorithm can build up such an data structure with no additional cost, which is an advantage compared to traditional mesh formats. This was also mentioned by Gumhold and Straßer [GS98].

As it can be seen in Table 5.5, I observed that my compression algorithm performs faster compared to GZIP with its highest compression level. The decompression of the mesh using my algorithm is only slightly faster than the compression. This is due to the symmetric compression applied in the different components of my algorithm. All of them doing approximately the same work for compression or decompression: The Cut-Border Machine has to build up the same cut-border for both cases, the attributes have to be predicted in both cases using the same algorithms and the Arithmetic Coder has to do the same interval reductions in both cases. The only optimization done for decompression is, that the upper interval limit has not to be computed for Arithmetic Coding because the interval renormalization is done by the input bits, instead of analysing the the bounds

of the interval. The decompression of the GZIP archive is however way faster than the compression of it because it is an asymmetric compressing approach.

5.2 Comparison against existing implementations

As described in Section 2.4 there are different fully featured compression tools: OpenCTM, Draco, Open3DGC and WebGL loader. Due to the inability of Open3DGC and webgl-loader to encode attributes losslessly, they cannot be tested against my implementation. Implementing other algorithms proposed in Chapter 2 would be beyond the scope of this thesis, so I only evaluate my implementation against OpenCTM and Draco. My implementation works on the compressed and then decompressed output of the specific implementation. Because OpenCTM and Draco are not able to encode arbitrary numbers of attributes, my algorithm achieves different rates for the same model.

	OpenCTM		Draco	
Model	r_{my}	r_{ctm}	r_{my}	r_{draco}
Bronze Akt	**11.86:1**	6.50:1	**5.28:1**	3.04:1
Tex Akt	**6.16:1**	5.79:1	**5.72:1**	3.07:1
Power Plant	7.05:1	**33.06:1**	**6.42:1**	3.12:1
Bunny	**8.96:1**	5.48:1	**4.63:1**	3.10:1
Mushroom	**10.95:1**	8.65:1	**12.53:1**	3.06:1

Table 5.6: Comparison with existing compressor implementations.

In Table 5.6 the compression rates of the different tools can be seen. OpenCTM uses LZMA, which is a dictionary coder like GZIP, that gives better results on the Power Plant model (33.06:1), as also mentioned in Section 5.1. For the other models, my algorithm gives better results with rates between 6.16:1 and 11.86:1, OpenCTM archives a rate between 5.48:1 and 8.65:1. A integration of GZIP into my algorithm could possibly give better results on the Power Plant model, as I mention in Section 6.1.

Draco compresses with rates of approximately 3.08:1. Compared to my algorithm, which achieves rates of between 4.63:1 to 12.53:1 for this selection of attributes, this is quite poor and due to their use of static Huffman Coding for connectivity data and bad prediction schemes for attributes which are not positions.

5.3 Graphical inspector

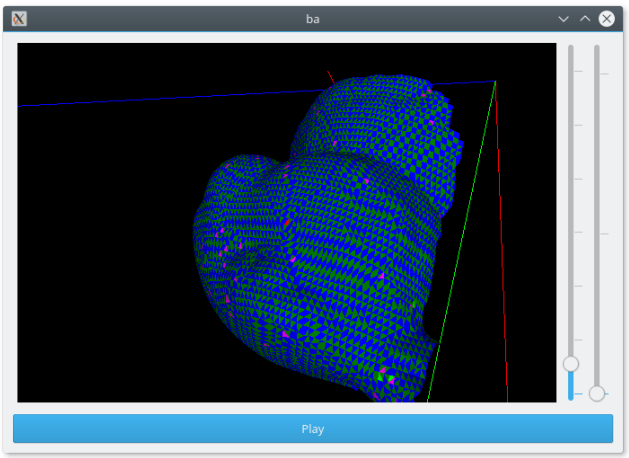


Figure 5.2: Graphical inspector window.

During implementation of the Cut-Border Machine, it turned out to be a very helpful to visualize the compression steps of the algorithm, which was also done by Gumhold [Gum99]. This was due to its very general description of Gumhold and Straßer [GS98] and to support analysing the algorithm for extending it to support non-manifold meshes. To visualize the algorithm, I implemented a simple OpenGL window, as it can be seen in Figure 5.2. It visualizes the triangles created by the Cut-Border Machine dynamically in different colors while the algorithm runs. It marks mesh borders and shows the triangle that is currently traversed by the Cut-Border Machine. It also allows to synthetically slow down and pause the algorithm to allow iterating the mesh step by step. Also, as it was shown in Section 4.2, it visualizes the differences between different traversal orders very clear (Figure 4.2).

6 Conclusion

In this thesis I presented a lossless compression algorithm that provides high compression rates on arbitrary triangle meshes. It archives compression rates of between 3.43:1 to 6.53:1 and compared to GZIP rates of between 0.82:1 and 1.80:1. It is well suited for real world models, i.e. 3D reconstructions, due to their smooth attributes, which can be well predicted but can also be used in any area of application due to the few assumptions on the input mesh. The algorithm performs fast enough to allow for compression in non real time areas, e.g. for archiving purposes or for transmission over rate or traffic limited networks. However, faster algorithms can be developed based on this thesis with a potential loss of compression efficiency, e.g. by replacing the Arithmetic Coder with a faster compression backend.

6.1 Future work

In this section, I would like to mention a couple of changes that can be applied to the presented compression algorithm in order to support arbitrary polygons and to achieve better compression speed or rates.

Arbitrary polygons In some areas, like CAD modeling, quads are used as well as triangles. This makes it useful to be be able to compress arbitrary polygons. For this the connectivity data structures must be extended to support arbitrary polygon meshes and the Cut-Border Machine must be extended as follows: When the gate reaches a polygon that is not a triangle the Cut-Border Machine must cut this polygon into a triangle fan in way that all triangles of the polygon are traversed consecutively. This is done by cutting the polygon into a triangle fan beginning from the gate towards the direction of the new gate, which results in consecutive traversal in case of depth-first traversal as mentioned in Section 4.2. Additionally, a number must be encoded showing how many triangles are included in this polygon to reconstruct the polygon successfully during decompression. This can be done by adding a symbol before every polygon. Thus pure triangle or

quad meshes will automatically omit this as the model only contains a single symbol.

Point clouds In many areas like 3D laser scanning or stereoscopic reconstruction algorithms, it would be useful to encode point clouds as well. A point cloud can be seen as a set of attributes with no connectivity. Compressing such data requires sorting of the points by a feasible criteria to apply delta coding of the attributes. Furthermore, the implementation of the attribute data structures presented in Section 4.4 can be integrated into this point cloud compressor.

Materials Some meshes can be grouped into different sub-meshes that have equal material properties for their triangles, e.g. texture maps. This requires the algorithm to compress these components separately with the disadvantage of losing of connectivity between them. To support different materials of a mesh, an efficient way must be found to restore connectivity between these components. This could possibly be done by connecting whole borders instead of connecting each vertices between the sub-meshes separately.

Predictions Some prediction schemes, I presented in Section 4.4 can be optimized in order to achieve better compression rates. To predict vertex attributes, different predictions schemes can be applied instead of relying on the parallelogram for all types of attributes. The current approach of predicting triangle-vertex attributes can be improved further by including adjacency data as well as history data.

Quantization In Section 5.1 I showed, that most of resulting compressed data accounts for the attributes. Since meshes often do not need to be compressed lossless, e.g. in the areas of web 3D rendering or video games, a loss of attribute precision is acceptable to gain faster transmission times. Therefore, an introduction of a quantization step would make this algorithm more efficient but still allows generality on meshes by keeping it optional.

Backend To gain more performance, the Arithmetic Coder could be replaced by Huffman Coding, which gives better compression speed but less compression rates. Alternatively, the recently developed ANS coder by Duda [Dud13] could be used as a replacement for the compression backend as it is done in the Draco project [Goo17], which could speed up the algorithm and compress with rates similar to the Arithmetic Coder. Unfortunately, the ANS coder has the disadvantage, that it is not easily integratable due

to its limitation of decompressing in reverse order. Another options is to use different GZIP streams for each attribute and the connectivity. This change requires to rethink the strategies used for conditional frequencies, which could possibly be replaced by further different GZIP streams for each condition.

Bibliography

[Eli55] Peter Elias. “Predictive coding–I.” In: *IRE Transactions on Information Theory* 1.1 (1955), pp. 16–24.

[RL79] Jorma Rissanen and Glen G Langdon. “Arithmetic coding.” In: *IBM Journal of research and development* 23.2 (1979), pp. 149–162.

[Fen93] Peter Fenwick. “A new data structure for cumulative probability tables.” In: *Software-Practice and Experience* (1993).

[Müc93] Ernst Peter Mücke. “Shapes and implementations in three-dimensional geometry.” In: (1993).

[Dee95] Michael Deering. “Geometry compression.” In: *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*. ACM. 1995, pp. 13–20.

[Fen95] P Fenwick. *A New Data sturcture for cumulative Probability Tables: an Improved Frequency to Symbol Algorithm*. Tech. rep. Department of Computer Science, The University of Auckland, New Zealand, 1995.

[Hop96] Hugues Hoppe. “Progressive meshes.” In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. ACM. 1996, pp. 99–108.

[CKS98] Swen Campagna, Leif Kobbelt, and Hans-Peter Seidel. “Directed edges—A scalable representation for triangle meshes.” In: *Journal of Graphics tools* 3.4 (1998), pp. 1–11.

[GS98] Stefan Gumhold and Wolfgang Straßer. “Real time compression of triangle mesh connectivity.” In: *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. ACM. 1998, pp. 133–140.

[MNW98] Alistair Moffat, Radford M Neal, and Ian H Witten. “Arithmetic coding revisited.” In: *ACM Transactions on Information Systems (TOIS)* 16.3 (1998), pp. 256–294.

[TG98] Costa Touma and Craig Gotsman. “Triangle Mesh Compression.” In: *Proceedings of the Graphics Interface 1998 Conference, June 18-20, 1998, Vancouver, BC, Canada*. June 1998, pp. 26–34. URL: http://graphicsinterface.org/wp-content/uploads/gi1998-4.pdf.

[Gum99] Stefan Gumhold. “Improved cut-border machine for triangle mesh compression.” In: *Erlangen Workshop*. Vol. 99. 1999, pp. 261–268.

[GGS99] Stefan Gumhold, Stefan Guthe, and Wolfgang Straßer. “Tetrahedral mesh compression with the cut-border machine.” In: *Proceedings of the conference on Visualization'99: celebrating ten years*. IEEE Computer Society Press. 1999, pp. 51–58.

[Ros99] Jarek Rossignac. “Edgebreaker: Connectivity compression for triangle meshes.” In: *IEEE transactions on visualization and computer graphics* 5.1 (1999), pp. 47–61.

[IGG01] Martin Isenburg, Stefan Gumhold, and Craig Gotsman. “Connectivity shapes.” In: *Proceedings of the conference on Visualization'01*. IEEE Computer Society. 2001, pp. 135–142.

[AG05] Pierre Alliez and Craig Gotsman. “Recent advances in compression of 3D meshes.” In: *Advances in multiresolution for geometric modelling*. Springer, 2005, pp. 3–26.

[PKK05] Jingliang Peng, Chang-Su Kim, and C-C Jay Kuo. “Technologies for 3D mesh compression: A survey.” In: *Journal of Visual Communication and Image Representation* 16.6 (2005), pp. 688–733.

[LI06] Peter Lindstrom and Martin Isenburg. “Fast and efficient compression of floating-point data.” In: *IEEE transactions on visualization and computer graphics* 12.5 (2006), pp. 1245–1250.

[MZP09] Khaled Mamou, Titus Zaharia, and Françoise Prêteux. “TFAN: A low complexity 3D mesh compression algorithm.” In: *Computer Animation and Virtual Worlds* 20.2-3 (2009), pp. 343–354.

[Gee10] Marcus Geelnard. *OpenCTM - Compression of 3D triangle meshes*. http://openctm.sourceforge.net/. Accessed: 2017-01-21. 2010.

[Goo11] Google. *webgl-loader*. https://code.google.com/archive/p/webgl-loader/. Accessed: 2017-01-31. 2011.

[AMD13] AMD. *Open 3D Graphics Compression (Open3DGC)*. https://github.com/amd/rest3d/tree/master/server/o3dgc. Accessed: 2017-01-31. 2013.

[Dud13] Jarek Duda. “Asymmetric numeral systems as close to capacity low state entropy coders.” In: *CoRR* abs/1311.2540 (2013). URL: http://arxiv.org/abs/1311.2540.

[Lim+13] Max Limper et al. “Fast delivery of 3D web content: a case study.” In: *Proceedings of the 18th International Conference on 3D Web Technology*. ACM. 2013, pp. 11–17.

[FLG14] Simon Fuhrmann, Fabian Langguth, and Michael Goesele. “MVE-A Multi-View Reconstruction Environment.” In: *GCH*. 2014, pp. 11–18.

[WMG14] Michael Waechter, Nils Moehrle, and Michael Goesele. “Let There Be Color! — Large-Scale Texturing of 3D Reconstructions.” In: *Proceedings of the European Conference on Computer Vision*. Springer, 2014.

[Mag+15] Adrien Maglo et al. “3d mesh compression: Survey, comparisons, and emerging trends.” In: *ACM Computing Surveys (CSUR)* 47.3 (2015), p. 44.

[Goo17] Google. *Introducing Draco: compression for 3D graphics*. https://opensource.googleblog.com/2017/01/introducing-draco-compression-for-3d.html. Accessed: 2017-01-21. 2017.